Optimization

Prof. Giuseppe Serra

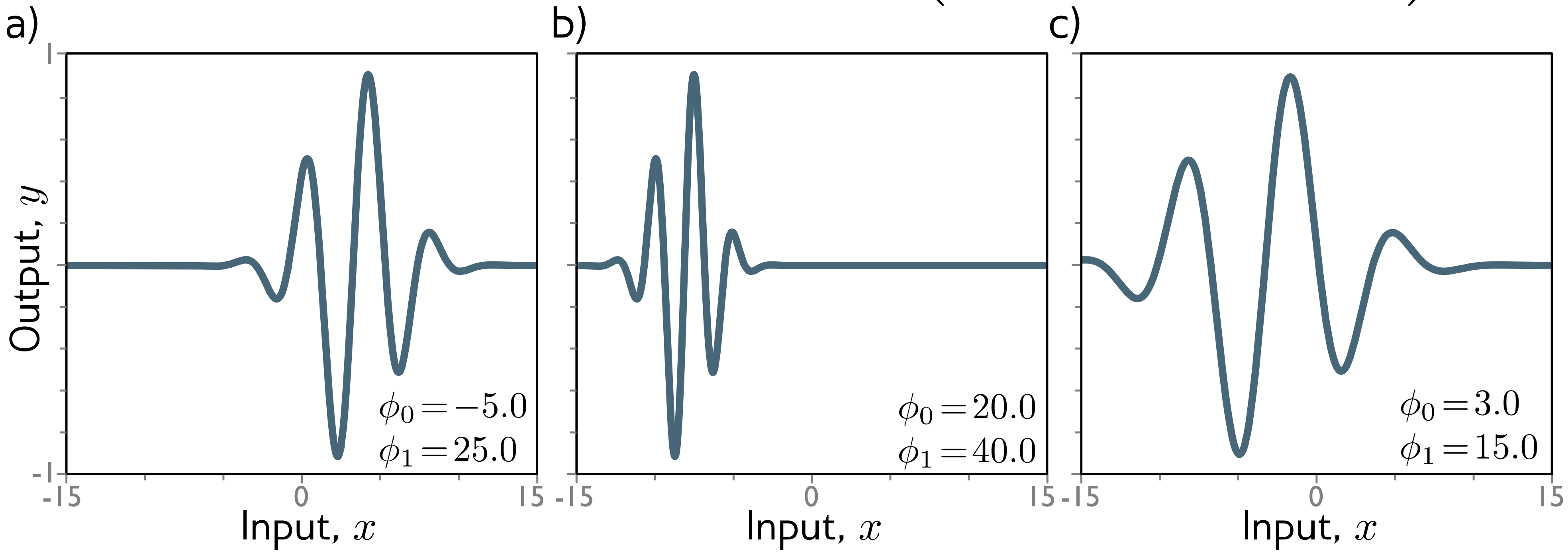
University of Udine

Gabor Model

Unfortunately, loss functions for most nonlinear models, including both shallow and deep networks, are non-convex
Visualizing neural network loss functions is challenging due to the number of parameters.

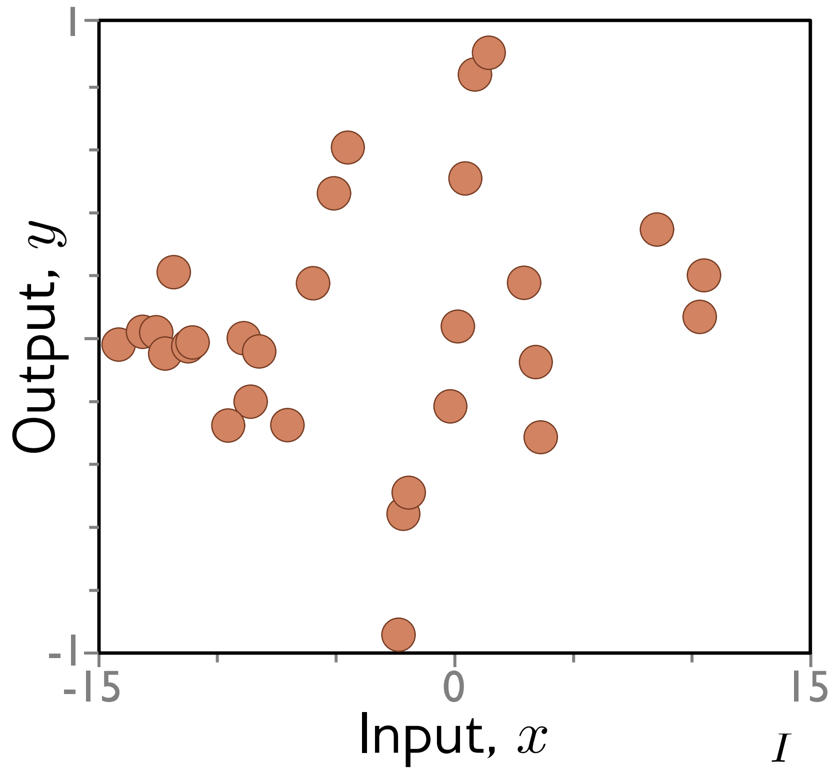
Let's explore a simple nonlinear function: Gabor Model

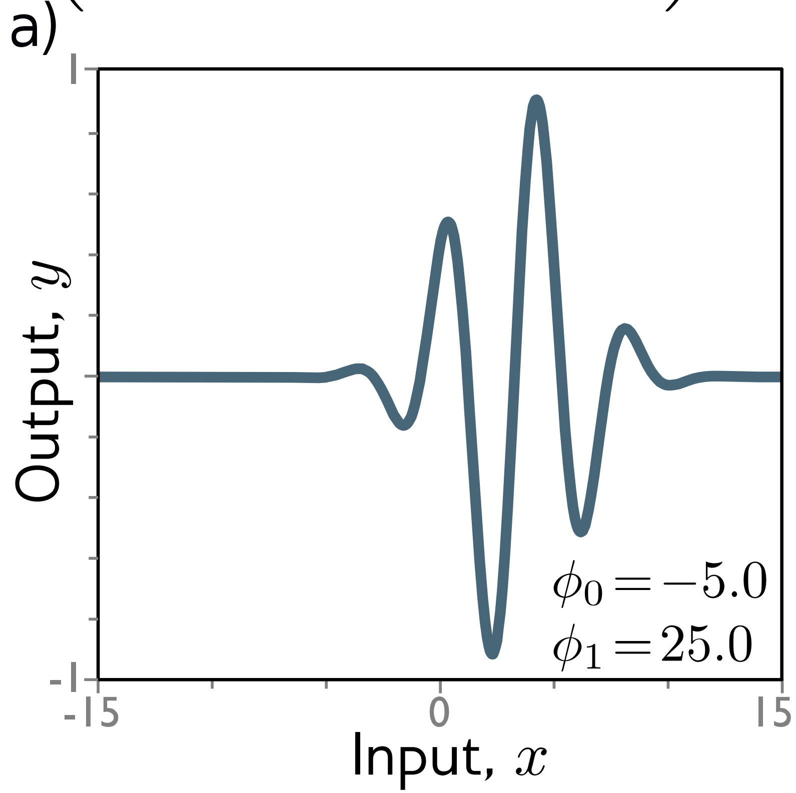
$$\mathrm{f}[x, \boldsymbol{\phi}] = \sin[\phi_0 + 0.06 \cdot \phi_1 x] \cdot \exp\left(-\frac{(\phi_0 + 0.06 \cdot \phi_1 x)^2}{8.0}\right)$$



Gabor Model - Training

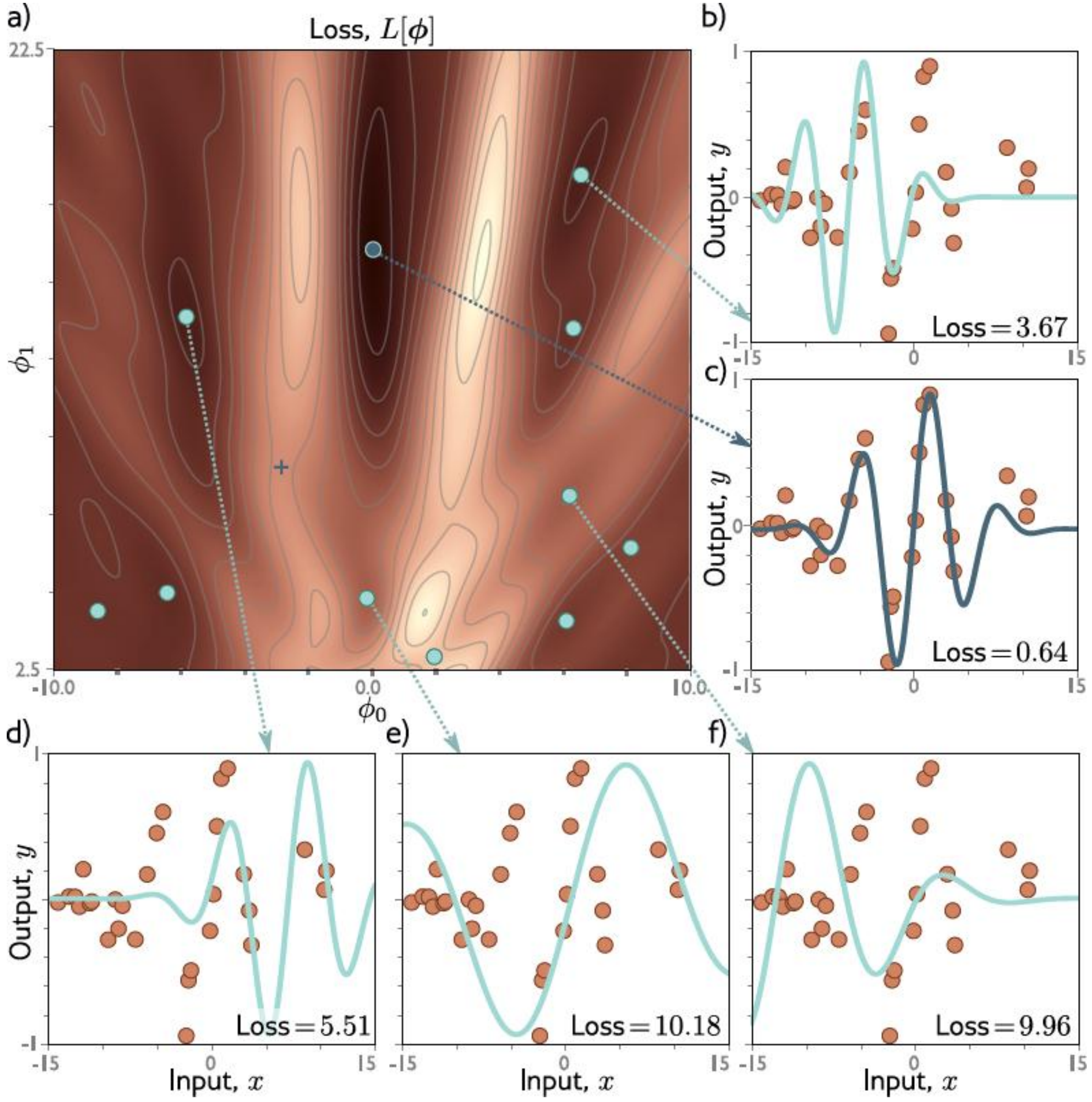
$$\mathrm{f}[x, \boldsymbol{\phi}] = \sin[\phi_0 + 0.06 \cdot \phi_1 x] \cdot \exp\left(-\frac{(\phi_0 + 0.06 \cdot \phi_1 x)^2}{8.0}\right)$$





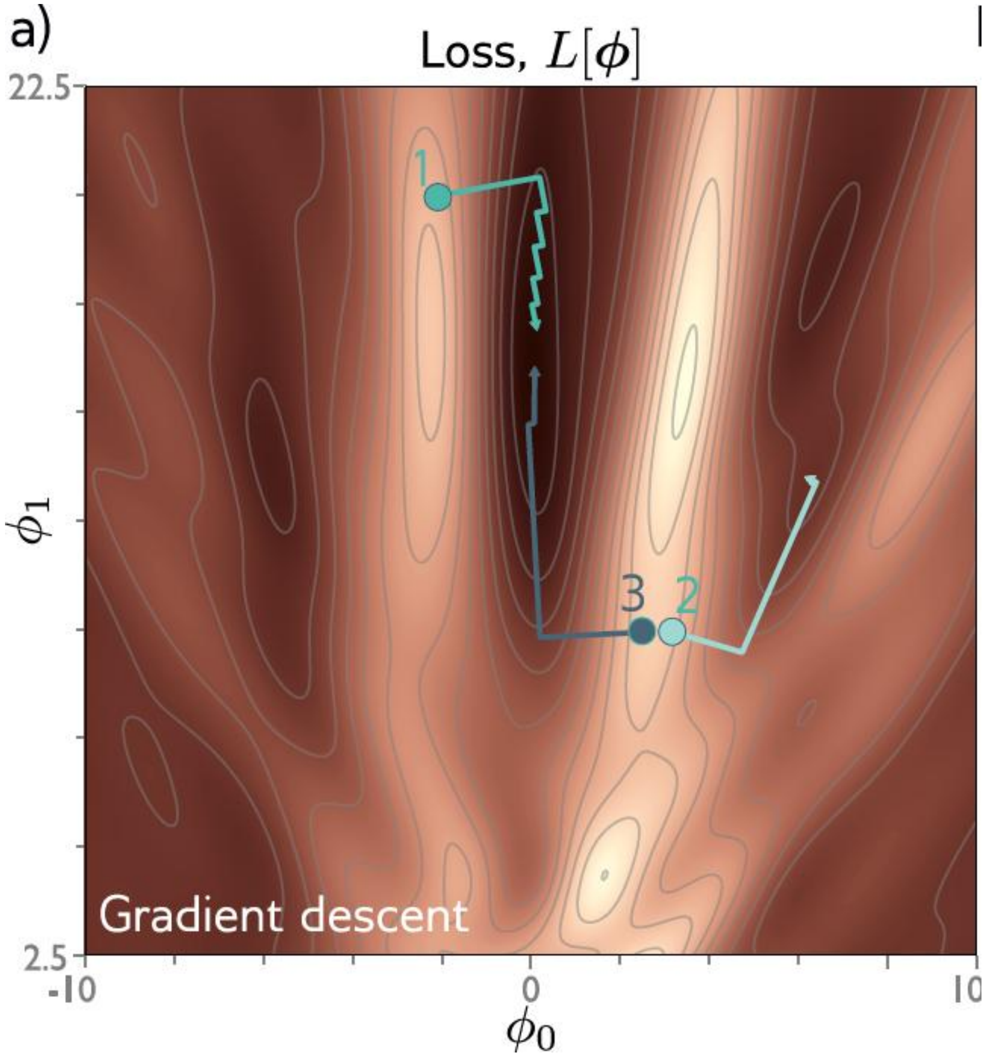
$$L[\boldsymbol{\phi}] = \sum_{i=1}^{I} (\mathrm{f}[x_i, \boldsymbol{\phi}] - y_i)^2$$

Gabor Model – Loss Function



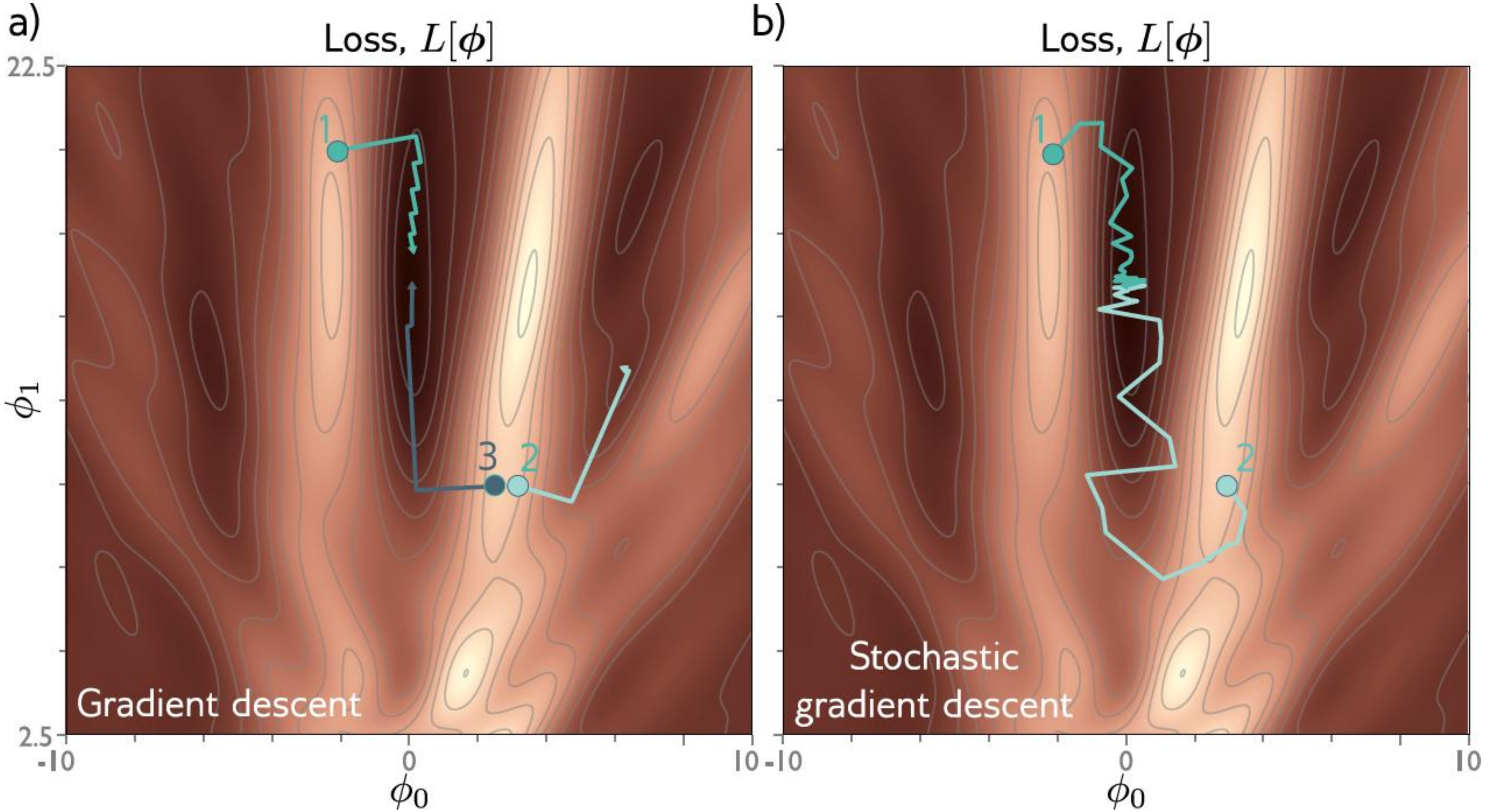
Gabor Model - Training

Gradient descent gets to the global minimum if we start in the right “valley”; Otherwise, descent to a local minimum; Or get stuck near a saddle point



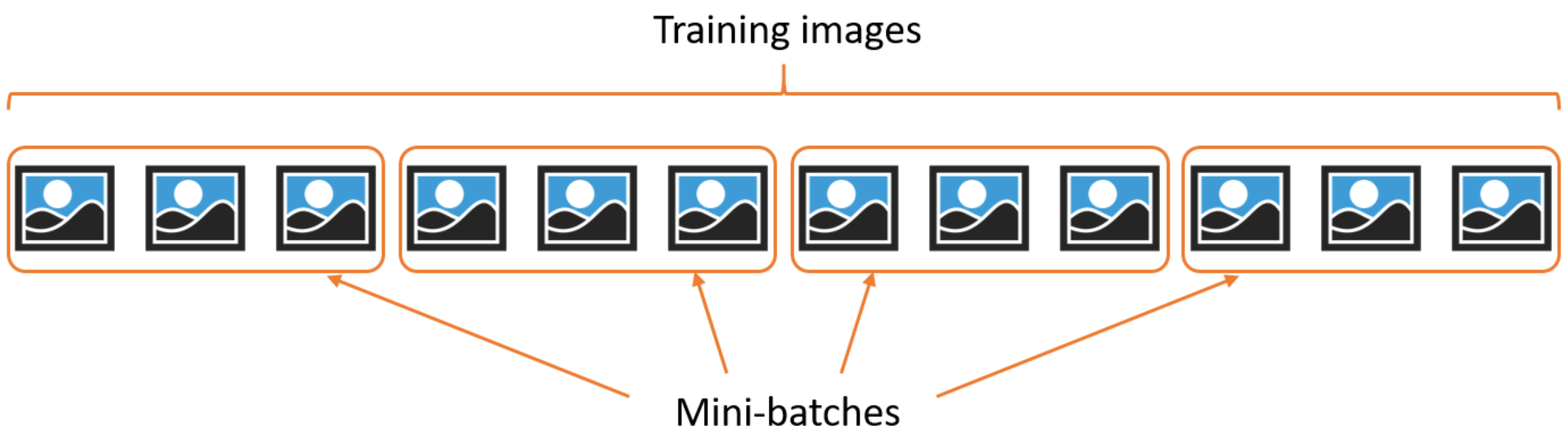
Gabor Model - Stochastic Gradient Descent

Compute gradient based on only a subset of points – a mini-batch; Work through dataset sampling without replacement; One pass though the data is called an epoch



Mini-Batches

Let's suppose you want to solve a Computer Vision task, and you have the following training set:



If you have small training set (e.g. 2000 samples), just use batch gradient descent

If you have larger training set typical mini-batch sizes are: 64, 128, 256, 512... 1024

The main constraint is the batch must be allocated in CPU/GPU memory. Therefore, it depends on your applications and your hardware.

Size of mini-batch can be defined as an hyperparameter. Thus, you need to test several values in order to find the correct one.

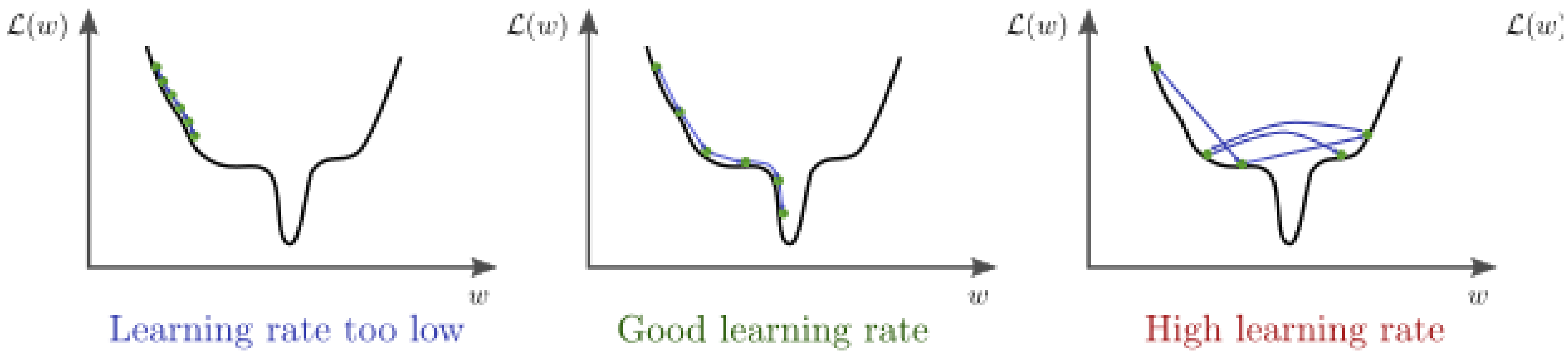
Stochastic Gradient Descent - Properties

- Can escape from local minima
- Adds noise, but still sensible updates as based on part of data
- Uses all data equally
- Less computationally expensive
- Seems to find better solutions

- Note: SGD is often applied with learning rate schedule. The learning rate starts at a high value and is decreased by a constant factor every N epochs.

Learning Rate Decay

As we saw, based on the learning rate we can have the following situation:



Learning Rate Decay

$$\alpha = \frac{1}{1 + decay - rate * epoch - num}\alpha_0$$

Example:

$\alpha_0 = 0.2; decay - rate = 1$

Epoch	α
1	0.1
2	0.067
3	0.05
4	0.04
…	…

Other formulas:

$\alpha = \beta^{epoch-num}\alpha_0$ with $\beta \in (0,1)$ - Exponentially Decay

$\alpha = \frac{k}{\sqrt{epoch-num}}\alpha_0$ with $k > 1$

Exponentially Weighted Average

First day: $\theta_1 = 4°$

Second day: $\theta_2 = 9°$ …

200° day: $\theta_{200} = 20°$ ….

The plot is very noisy.

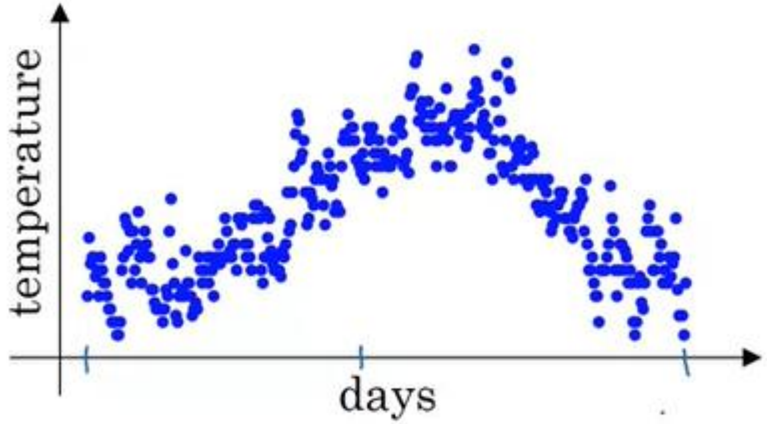
If you set

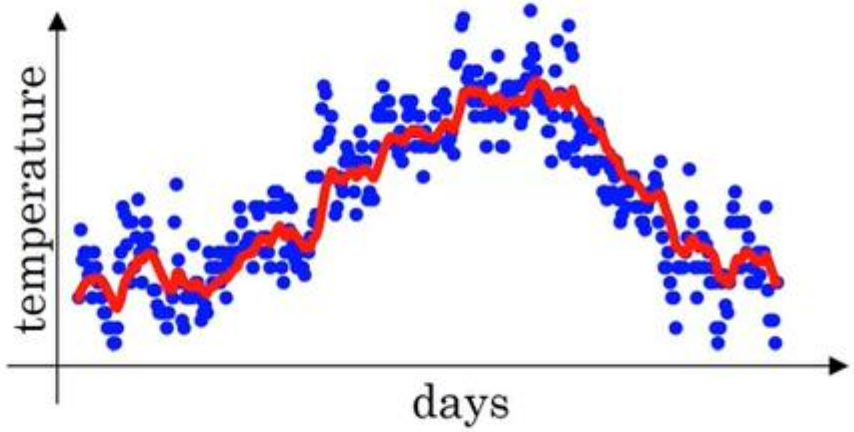
$v_0 = 0$

$v_1 = 0.9v_0 + 0.1\ \theta_1$

The read line is more smoothed curve

It is called **Exponentially Weighted Average** of the daily temperature





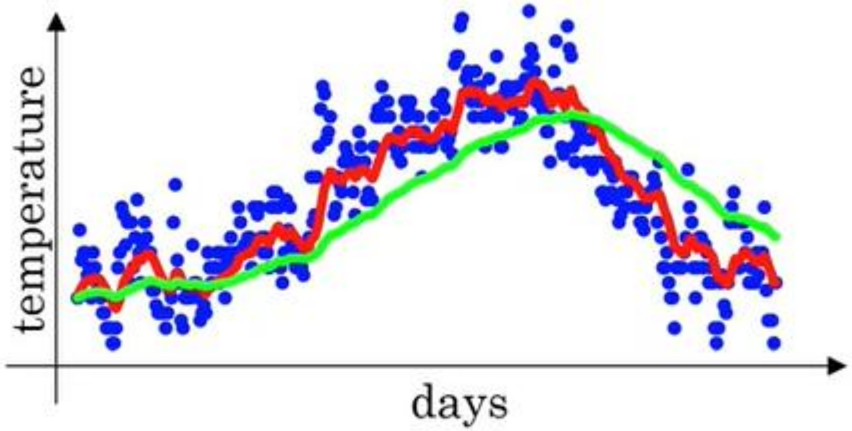
Exponentially Weighted Averages

$V_t = \beta V_{t-1} + (1-\beta)\theta_t$

It can be proved V_t is an approximation of $\frac{1}{1-\beta}$ days' temperatures

Red line: $\beta = 0.9$ (average of ≈ 10 days' temperature)

Green line $\beta = 0.98$ (average of ≈ 50 days' temperature)



Exponentially Weighted Averages

$\boldsymbol{V_t = \beta V_{t-1} + (1-\beta)\theta_t}$

Let's set $\beta = 0.9$ and $t = 100$

$V_{100} = 0.9V_{99} + 0.1\theta_{100}$

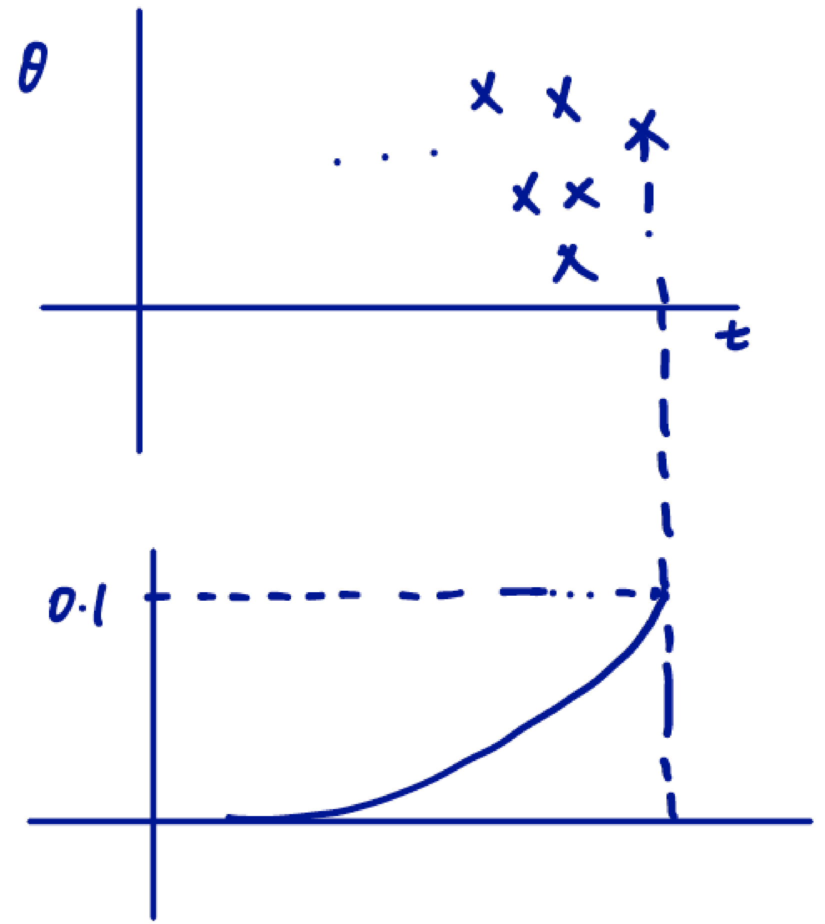
$V_{99} = 0.9V_{98} + 0.1\theta_{99}$

$V_{98} = 0.9V_{97} + 0.1\theta_{98}$

$\rightarrow \boldsymbol{V_{100}} = 0.1\theta_{100} + 0.9(0.9V_{98} + 0.1\theta_{99}) = 0.1\theta_{100} + 0.1 * 0.9\theta_{99} + (0.9)^2V_{98} =$

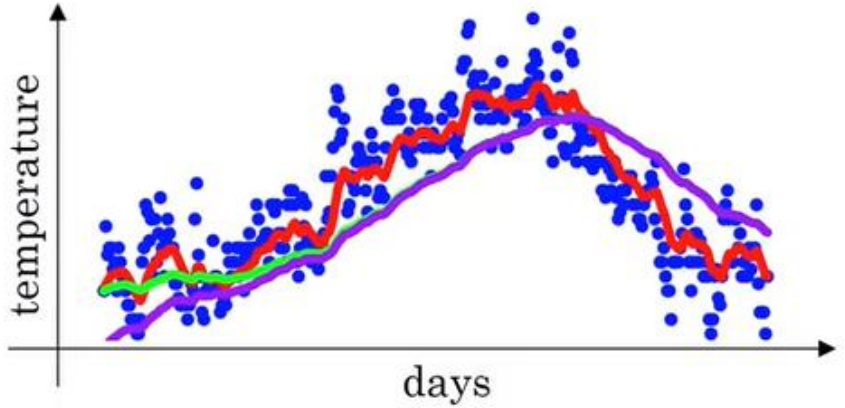
$= 0.1\theta_{100} + 0.1 * 0.9\theta_{99} + (0.9)^2(0.9V_{97} + 0.1\theta_{98})$

$= \boldsymbol{0.1\theta_{100} + 0.1 * 0.9\theta_{99} + 0.1 * (0.9)^2\theta_{98} + (0.9)^3V_{97}}$ ....



Bias Correction

If you implement this formula $V_t = \beta V_{t-1} + (1-\beta)\theta_t$ you will have the purple curve (it's not the best one)



Let's see an example:

If you set $\beta = 0.98$ and $\boldsymbol{V_0} = \mathbf{0}$

$\rightarrow \mathbf{V_1} = 0.98V_0 + 0.02\theta_1 = \mathbf{V_1} = \mathbf{0.02}\boldsymbol{\theta_1}$

$\rightarrow \mathbf{V_2} = 0.98V_1 + 0.02\theta_2 = 0.98 * 0.02 * \theta_1 + 0.02\theta_2 = \mathbf{0.0196}\boldsymbol{\theta_1} + 0.02\theta_2$

Note: V_1 and V_2 are not very good estimations of the first two days of temperature of the year.

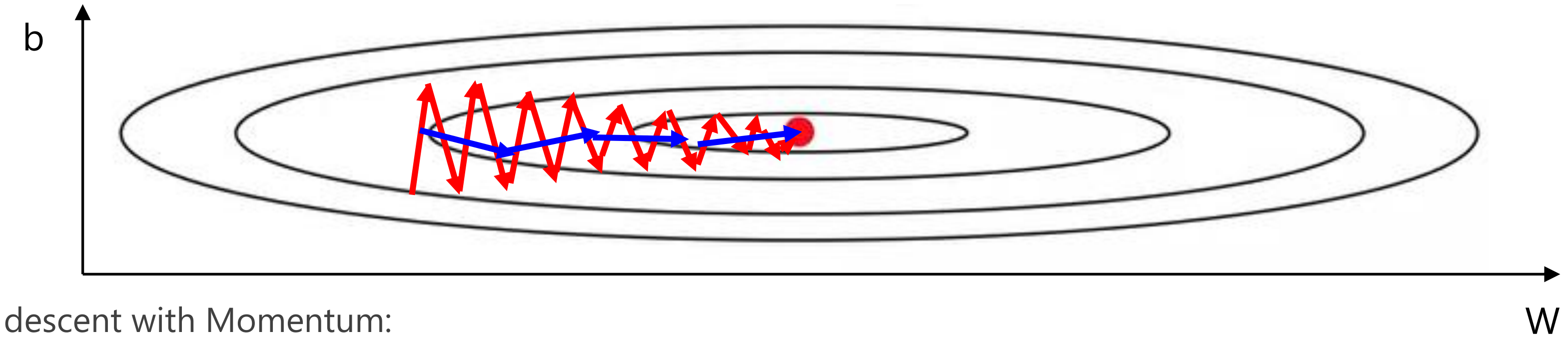
Thus, we can add a bias term. The revised formula is: $\tilde{V}_t = \frac{V_t}{1-\beta^t}$ (green curve)

$$\rightarrow t = 2 \ \rightarrow 1 - \beta^t = 1 - (0.98)^2 = 0.0396 \ \rightarrow \ \tilde{V}_2 = \frac{0.0196\theta_1 + 0.02\theta_2}{0.0396}$$

Note: when $t \rightarrow \infty$ the term $1 - \beta^t \rightarrow 1$

Gradient Decent with Momentum

Gradient Descent (red line):



Gradient descent with Momentum:

For each iteration compute $\frac{d\mathcal{L}}{dW}$ and $\frac{d\mathcal{L}}{db}$

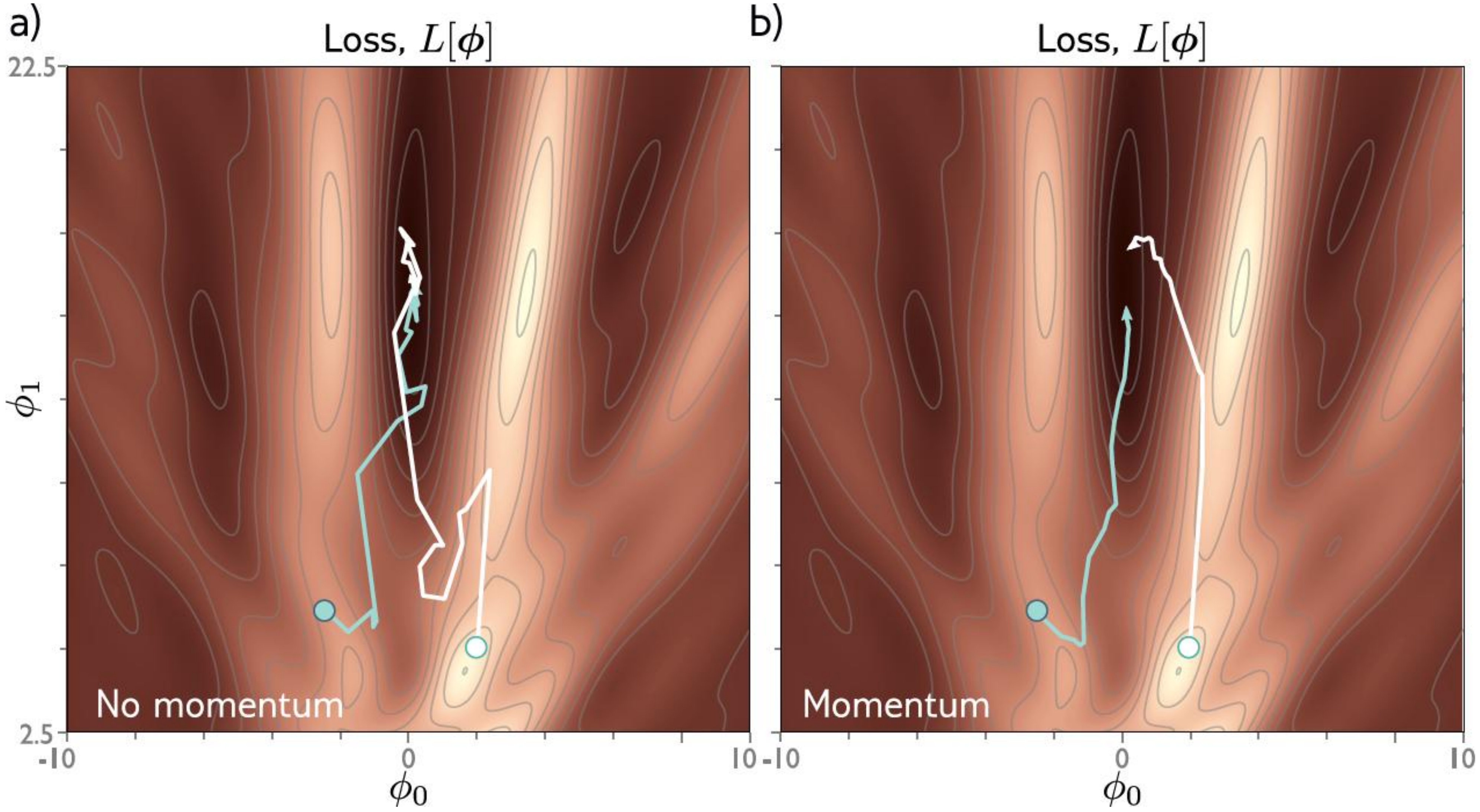
$V_{dw} = \beta V_{dw} + (1-\beta)\,\frac{d\mathcal{L}}{dW};$ $\quad V_{db} = \beta V_{db} + (1-\beta)\frac{d\mathcal{L}}{db}$ good value of β=0.9

$W := W - \alpha\ V_{dw};$ $\quad$ b $:= b - \alpha\ V_{db}$

when considering "Exponentially Weighted Averages" in this case (see figure), the average of b is more or less "constant" (vertical direction), while W goes toward to right

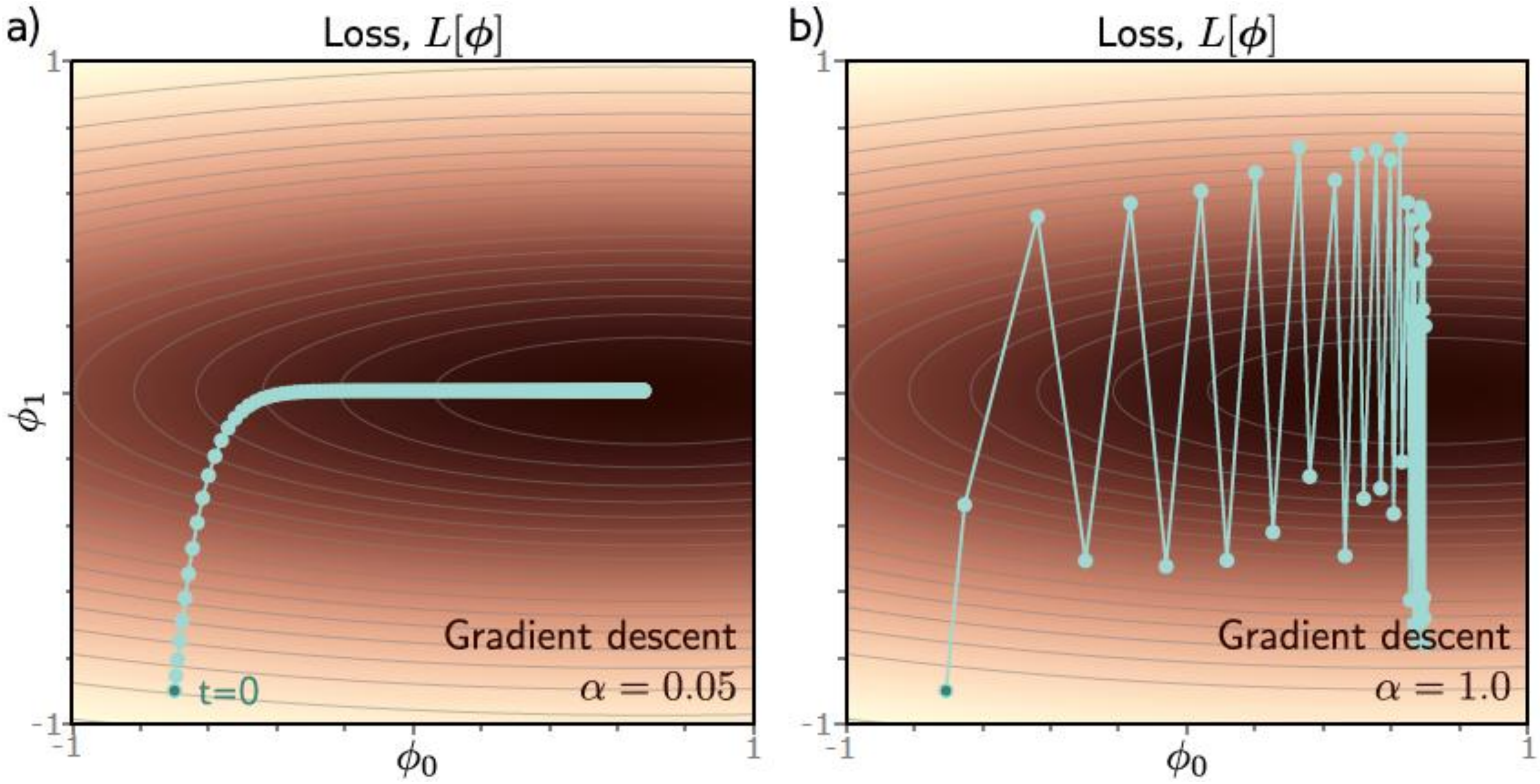
Gradient descent with Momentum almost always works empirically faster than standard gradient descent algorithm.

Gradient Decent with Momentum



Gradient Descent and Learning Rate

In this case if the learning rate is small, the algorithm takes a long time to reach the minimum; if the learning rate is large the algorithm becomes instable.

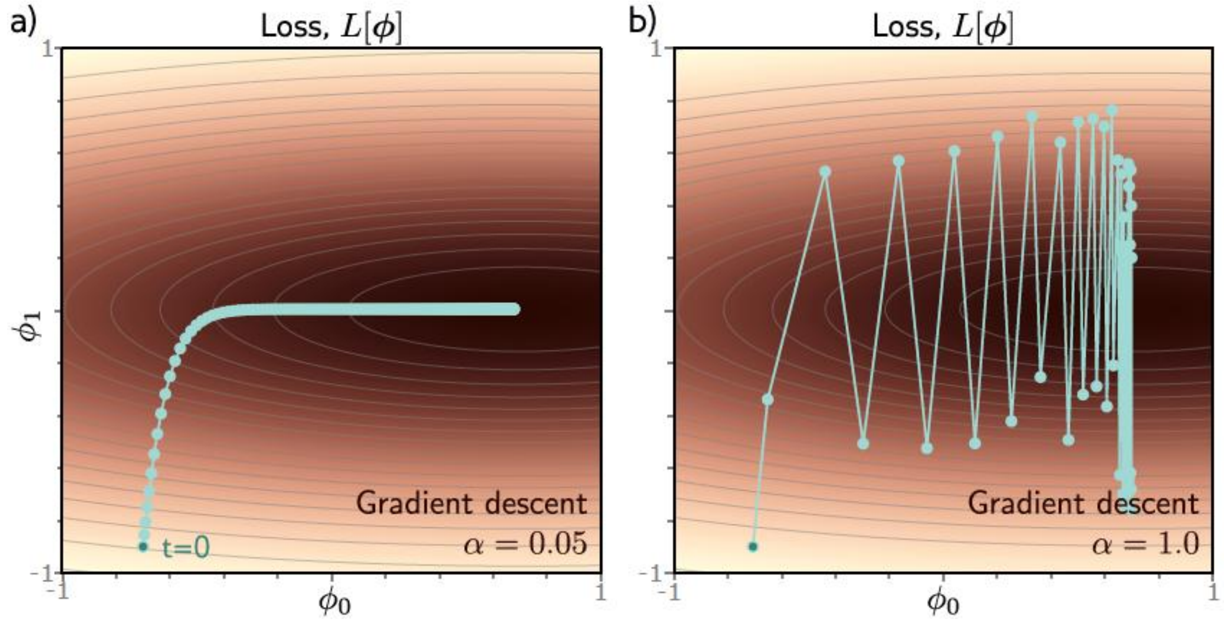


Gradient Descent and Learning Rate

Gradient descent with a fixed step size has the following undesirable property:

- it makes large adjustments to parameters associated with large gradients (where perhaps we should be more cautious) and small adjustments to parameters associated with small gradients (where perhaps we should explore further).

- When the gradient of the loss surface is much steeper in one direction than another, it is difficult to choose a learning rate that (i) makes good progress in both directions and (ii) is stable (figures 6.9a–b).



Normalized gradients

Measure mean and pointwise squared gradient

$$\mathbf{m}_{t+1} \leftarrow \frac{\partial L[\phi_t]}{\partial \phi}$$

$$\mathbf{v}_{t+1} \leftarrow \frac{\partial L[\phi_t]}{\partial \phi}^2$$

Normalize:

$$\phi_{t+1} \leftarrow \phi_t - \alpha \cdot \frac{\mathbf{m}_{t+1}}{\sqrt{\mathbf{v}_{t+1}} + \epsilon}$$

Normalized gradients

Measure mean and pointwise squared gradient

$$\mathbf{m}_{t+1} \leftarrow \frac{\partial L[\phi_t]}{\partial \phi}$$

$$\mathbf{v}_{t+1} \leftarrow \frac{\partial L[\phi_t]}{\partial \phi}^2$$

$$\mathbf{m}_{t+1} = \begin{bmatrix} 3.0 \\ -2.0 \\ 5.0 \end{bmatrix}$$

$$\mathbf{v}_{t+1} = \begin{bmatrix} 9.0 \\ 4.0 \\ 25.0 \end{bmatrix}$$

Normalize:

$$\phi_{t+1} \leftarrow \phi_t - \alpha \cdot \frac{\mathbf{m}_{t+1}}{\sqrt{\mathbf{v}_{t+1}} + \epsilon}$$

$$\frac{\mathbf{m}_{t+1}}{\sqrt{\mathbf{v}_{t+1}} + \epsilon} = \begin{bmatrix} 1.0 \\ -1.0 \\ 1.0 \end{bmatrix}$$

Normalized gradients

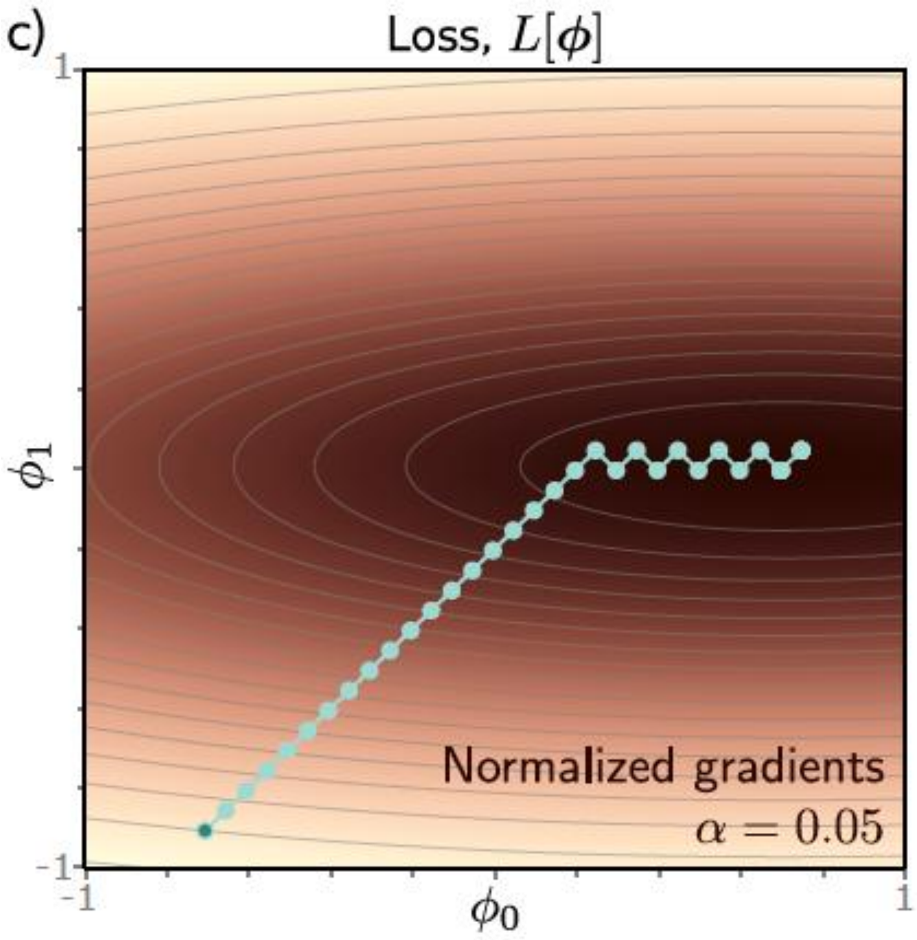
Measure mean and pointwise squared gradient

$$\mathbf{m}_{t+1} \leftarrow \frac{\partial L[\phi_t]}{\partial \phi}$$

$$\mathbf{v}_{t+1} \leftarrow \frac{\partial L[\phi_t]}{\partial \phi}^2$$

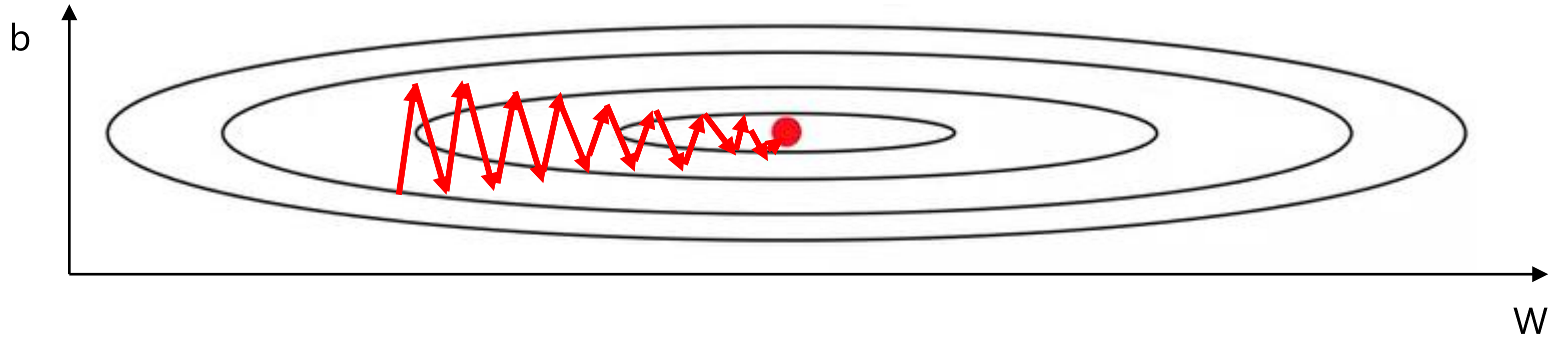
Normalize:

$$\phi_{t+1} \leftarrow \phi_t - \alpha \cdot \frac{\mathbf{m}_{t+1}}{\sqrt{\mathbf{v}_{t+1}} + \epsilon}$$



RMSprop

Gradient Descent (red line):



RMSprop:

For each iteration compute $\frac{d\mathcal{L}}{dW}$ and $\frac{d\mathcal{L}}{db}$

$S_{dw} = \beta S_{dw} + (1-\beta)\left(\frac{d\mathcal{L}}{dW}\right)^2$; $\quad S_{db} = \beta S_{db} + (1-\beta)\left(\frac{d\mathcal{L}}{db}\right)^2 \quad$ good value of β=0.9

$W := W - \alpha \frac{d\mathcal{L}}{dW} / (\sqrt{S_{dw}} + \epsilon)$; $\quad$ b $:= b - \alpha \frac{d\mathcal{L}}{db} / (\sqrt{S_{db}} + \epsilon) \quad$ good value $\epsilon = 10^{-8}$ (it is just added to avoiding division by zero)

In our example $\frac{d\mathcal{L}}{dW}$ are a small values, while $\frac{d\mathcal{L}}{db}$ are large values. Therefore, when you update b you divide for a large number the derivative, so the oscillations will be reduced.

Adaptive Moment Estimation Optimizer (ADAM)

For iteration compute $\frac{d\mathcal{L}}{dW}$ and $\frac{d\mathcal{L}}{db}$

$V_{dw} = \beta_1 V_{dw} + (1-\beta_1)\frac{d\mathcal{L}}{dW}$; $\quad V_{db} = \beta_1 V_{db} + (1-\beta_1)\frac{d\mathcal{L}}{db}$

$S_{dw} = \beta_2 S_{dw} + (1-\beta_2)\left(\frac{d\mathcal{L}}{dW}\right)^2$; $\quad S_{db} = \beta_2 S_{db} + (1-\beta_2)\left(\frac{d\mathcal{L}}{db}\right)^2$

$\tilde{V}_{dw} = \frac{V_{dw}}{(1-\beta_1^t)}$ $\quad \tilde{V}_{db} = \frac{V_{db}}{(1-\beta_1^t)}$

$\tilde{S}_{dw} = \frac{S_{dw}}{(1-\beta_2^t)}$ $\quad \tilde{S}_{db} = \frac{S_{db}}{(1-\beta_2^t)}$

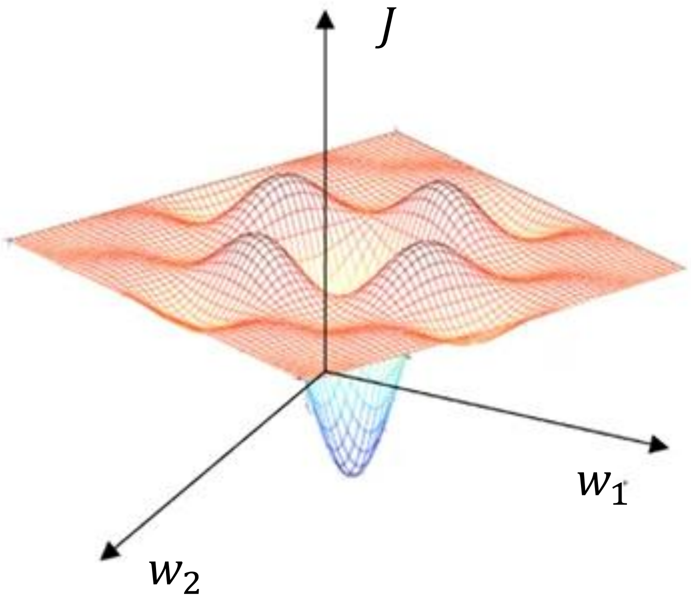
$W \coloneqq W - \alpha\, \tilde{V}_{dw}/(\sqrt{\tilde{S}_{dw}} + \epsilon)$; $\quad$ b $\coloneqq b - \alpha\, \tilde{V}_{db}/(\sqrt{\tilde{S}_{db}} + \epsilon)$

Hyperparameters choice:

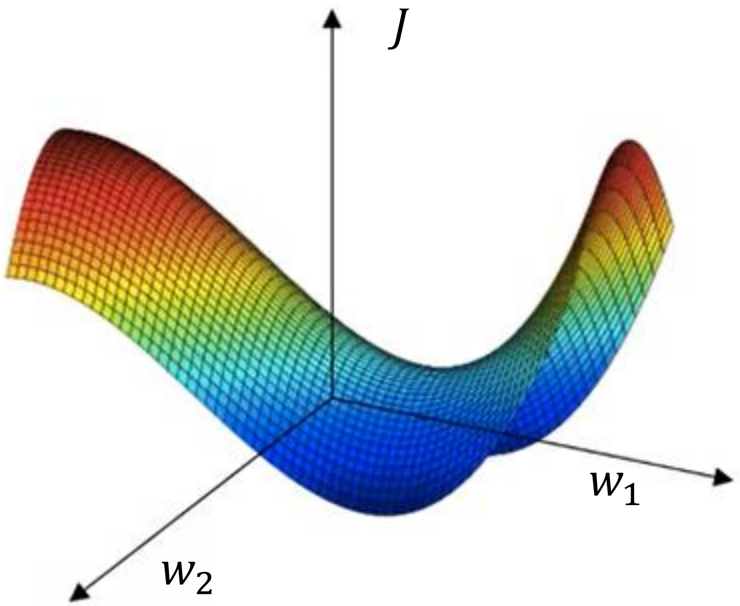
α: needs to be tune; β_1=0.9 (usual value); β_2=0.999 (usual value); $\epsilon = 10^{-8}$

Local Optima in Neural Networks

In Neural Network (we are working with high-dimensional space) the cost function likely has saddle points (figure B) instead of local point (figure A), because in **high-dimensional space it's unlikely to have all the dimensions like this:** ∪



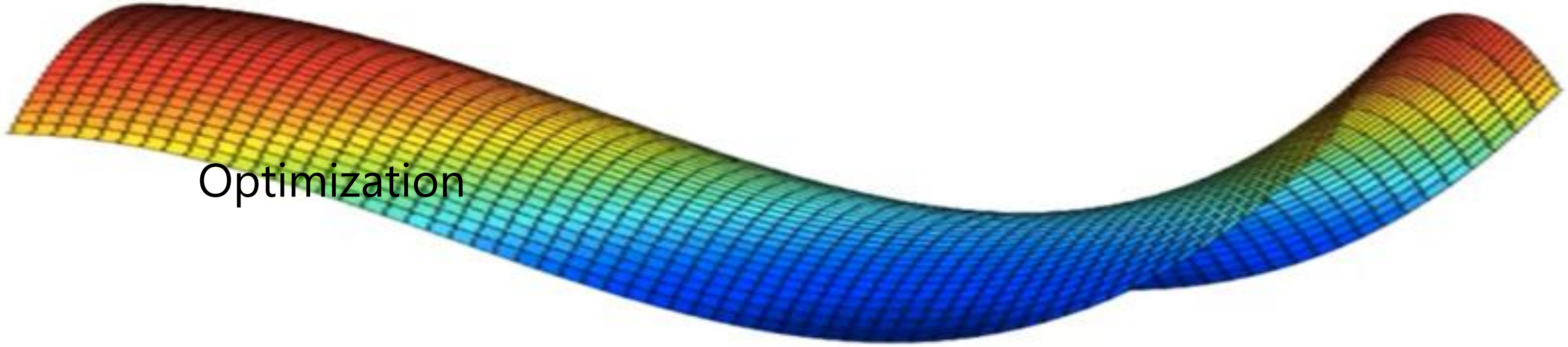
A



B

Problem of Plateaus

In this scenario Gradient Descent with momentum, RMSprop or Adam work better then gradient Descent as we saw before (see previous contour plots).



Suggested Readings

Simon J.D. Price “Understanding Deep Learning”, Chapters 6 (https://udlbook.github.io/udlbook/)